A constructive approach to testing model transformations

Camillo Fiorentini,[1] Alberto Momigliano,[1]
Mario Ornaghi,[1] and Iman Poernomo[2]

[1] Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy
{fiorenti,momiglia,ornaghi}@dsi.unimi.it
[2] Department of Computer Science, King's College London, Strand, London WC2R2LS, UK
iman.poernomo@kcl.ac.uk

Abstract. This paper concerns a formal encoding of the Object Management Group's Complete Meta-Object Facility (CMOF) in order to provide a more trustworthy software development lifecycle for Model Driven Architecture (MDA). We show how a form of constructive logic can be used to provide a uniform semantics of metamodels, model transformation specifications, model transformations and black-box *transformation tests*. A model's instantiation of a metamodel within the MOF is treated using the logic's realizability relationship, a kind of type inhabitation relationship that is expressive enough to characterize constraint conformance between terms and types. These notions enable us to formalize the notion of a correct model instantiation of a metamodel with constraints. We then adapt previous work on snapshot generation to generate input models from source metamodel specification with the purpose of testing model transformations.

1 Introduction

While model transformations have the potential to radically change the way we write code, the actual development of transformations themselves should be conducted according to standard software engineering principles. That is, transformations need to be either certified via some kind of formal method or else developed within the software development lifecycle. Currently the latter is the norm and, consequently, effective testing techniques are essential.

However, as observed in [3], the field currently lacks adequate techniques to support model transformation testing: testing techniques for code do not immediately carry across to the model transformation context, due to the complexity of the data under consideration [7]. In fact, test case data (test models) consists of metaobjects that conform to a given metamodel, while satisfying the precondition of the transformation's specification and additional constraints employed to target particular aspects of the implementation's capability. A metamodel itself has a particular structural semantics, consisting of a range of different constraints over a graph of associated metaclasses. It is therefore a non-trivial task to automatically generate a suitable range of instantiating metaobjects as test data.

The subject of this paper is a uniform framework for treating metamodels, model transformation specification and the automation of test case generation of data for black-box testing of model transformations to validate their adherence to given specifications. We argue that such a uniform treatment is necessary for ensuring trusted testing of model transformations. Transformations are powerful and, consequently, dangerous when wrong:

this is due to the systematic, potentially exponential, range of errors that a single bug in a transformation can introduce into generated code. But this danger can be counter-acted by effective model transformation testing, which demands that test models 1) are actual instances of their classifying metamodels and 2) satisfy the transformation specification preconditions and the tester's given contractual constraints for generation. In a standard approach, a number of different languages and systems may be involved: the MOF for defining metamodel types and constraints, a model transformation specification and implementation language and a system for generating test models that meet demands 1) and 2) above. Some of these paradigms may be conflated, but in general there will be some approaches with an inherent semantic gap. For example, one may employ Kermeta to define transformations and their specification, with MOF metamodels imported separately: a test case generation framework might be written in the former language, but would have its own separate semantics for understanding metamodel instantiation and the meaning of a transformation specification. As a result of employing semantically distinct languages and systems, one may end up without a formal guarantee that the generation approach actually does what it is meant to do.

This paper offers such a formal guarantee by employing a uniform formalism for representing MOF-based models and metamodels and their interrelationship (Section 2), model transformation specification and implementation and test-case generation (Section 3). We employ *constructive* logic to do this, since this formalism is inherently endowed with the ability to treat (meta)data, functions and their logical properties uniformly. Our encoding in logic is inspired by the relationship between models-as-terms and metamodels-as-types introduced in [14] following the Curry-Howard paradigm. The logic language corresponds to the metalevel, i.e., metamodels are represented by special logical formulae that satisfy the following property: if F_M is the formula representing a metamodel M, then terms of type F_M, which we call *information terms*, are encodings of the (valid) metamodel instances of M. Thus test case generation can be seen as information terms generation and the transformations themselves are specified as relationships between those terms.

Our work adheres to the transformation development approach of Jezequel et al., where design-by-contract and testing are used as a means of increasing trust [17]. The idea is that a transformation from models in a *source* language *SL* into model in a *target* language *TL* is equipped with a contract, consisting of a pre-condition and a post-condition. The transformation is tested with a suitable data set, consisting of a range of source models that satisfy the pre-condition, to ensure that it always yield target models that satisfy the post-condition. If we produce an input model that violates the post-condition, then the contract is not satisfied by the transformation and the transformation needs to be corrected. Our view of black-box testing of model transformations follows the framework given in Fig. 1, where a transformation Tr takes a source model *SM* as input, written in a source modelling language *SL*, and outputs a target model TM, written in a target language *TL*. The transformation is defined as a general mapping from elements of the *SL* that satisfy the pre-condition to elements of the *TL* that are required to satisfy the post-condition. A test case generator produces an appropriate set of source models: the transformation can then be tested by checking that each of the resulting target models preserve the contract (more precisely, are consistent with it). In our uniform approach we use the contract itself as an oracle. This does not entail, in general, that the contract can

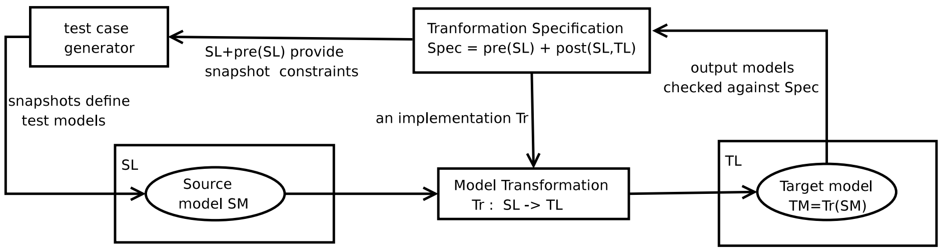

test case generator
SL+pre(SL) provide snapshot constraints
Tranformation Specification Spec = pre(SL) + post(SL,TL)
output models checked against Spec
snapshots define test models
an implementation Tr
SL
Source model SM
Model Transformation Tr : SL -> TL
TL
Target model TM=Tr(SM)

Fig. 1. Framework for testing transformation against contractual specifications

be used to perform the transformation. For example, it might express a loose relation, admitting a family of consistent implementations.

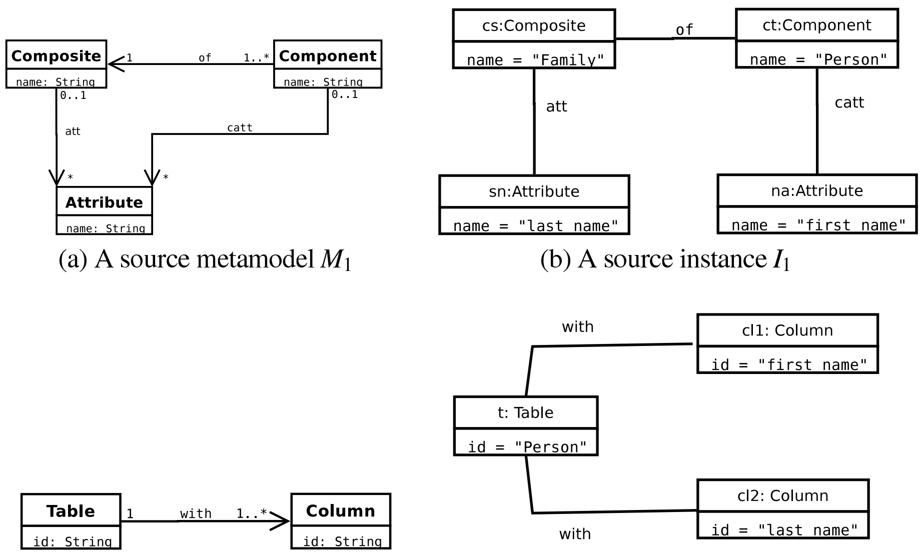
This paper thus proposes a solution to the open problem of how to derive in a fully declarative way an appropriate data set of source models for a given source language, pre-condition, post-condition and transformation. We will show how constructive logic enables a form of *meta-object snapshot generation* to solve this problem. Our approach is inspired by ordinary object-oriented snapshot generation [13], where objects are generated to satisfy given class specifications, for analogous same validation purposes.

2 A constructive encoding of the MOF

Model transformation test cases are generated according to constraints given over MOF-based metamodels. Our approach takes advantage of an approach to the uniform representation of both metamodel structure and semantics in relying on constructive logic, after the fashion of CooML's approach [6,13] and the constructive encoding of the earlier MOF in [14]. This constructive encoding has been shown to have a number of advantages as a formalisation of the MOF. In particular, realizability semantics can naturally treat metamodel instantiation, where classifiers are considered as instances of other classifiers: a feature that is prominent in the MOF itself (model instances are classification schemes, but themselves instantiate a metamodel scheme). This section sketches the principle of the constructive encoding, showing how the structure of metamodels can be understood as set theoretic signatures with a constructive *realizability* semantics to formalize instantiation. This realizability semantics will then be exploited in the next sections for test generation. The implication of our final result will be that the encoding presented here is a uniform framework for both reasoning about models and metamodels, for writing model transformations and also for generating test cases.

A metamodel for a modelling language has a definition as a collection of associated MOF metaclasses. For example, the full UML specification, like all OMG standards, has been defined in the CMOF. In Example 1 we present a very simple example of metamodel transformation, which will be used through the paper.

Example 1. Fig. 2 (a) and (c) shows two metamodels M_1 and M_2. The metamodel instances of M_1 represent models for simple composite structures, those of M_2 for tables. We are looking for a transformation of the `Component` meta-objects *ct* into `Table` meta-objects *t* `with` columns corresponding to the attributes linked to *ct* and to the composite containing *ct*.[1] For example, the metamodel instance I_2 in Fig. 2(d), modelling a `Person` according to the `Table` metamodel, corresponds to I_1 in Fig. 2(b), modelling a `Family` according to the `Composite` metamodel. Beside the multiplicity constraints shown in the meta-models, other constraints regard the `name` and `id` meta-attributes. Those will be discussed in Section 3.



(a) A source metamodel M_1 (b) A source instance I_1

(c) A target metamodel M_2 (d) A target instance I_2

Fig. 2. A source and a target metamodel

2.1 Encoding the structure of metamodels

We now describe a simple set-based encoding of the *structure* of metamodels and metaobject instantiations. Once we have established this encoding, we will turn our consideration to *semantics* and constraint conformance, using a constructive logic formalism.

Since we are only concerned with creating metaobject test cases for transformations, we need not deal with representing methods within class types and, for reasons of space, we consider the subset of the CMOF that permits a UML class-style representation of metamodel grammars.

Definition 1 (Signature for metamodel). *Assume* Class, Association, AssociationEnd *and* Property *are the meta-metaclasses for metaclasses, associations, association ends and properties in the CMOF. Take any metamodel M that consists of a set of (possibly*

[1] This is inspired by the `UML2RDB` challenge transformation proposed in [4], for whose details we refer to *op. cit.*

associated) metaclasses:

$$M : Set(\mathsf{Class}) \times Set(\mathsf{Association}) \times Set(\mathsf{AssociationEnd}) \times Set(\mathsf{Property})$$

Let M_{Class} *denote* $Set(\mathsf{Class})$*,* $M_{\mathsf{Association}}$ *denote* $Set(\mathsf{Association})$ *and so on. Then, the* signature $Sig(M)$ *for* M *is defined as* $\langle \mathsf{Sort}_M, \mathsf{Rel}_M, \mathsf{Op}_M \rangle$*, where* Sort_M *is a set of sort names,*[2] Rel_M *is a set of relations,* Op_M *is a set of sorted operations and defined to be the minimal tuple of sets satisfying the following conditions:*

- $\{T_C \mid C \in M_{\mathsf{Class}}\} \subseteq \mathsf{Sort}_M$*, where* T_C *is a unique sort name corresponding to a metaclass* $C \in M_{\mathsf{Class}}$*.*
- *Every data type* T *used within a* Property *of any* $C \in M_{\mathsf{Class}}$ *or* $A \in M_{\mathsf{Association}}$ *is taken as a sort in the signature:* $T \in \mathsf{Sort}_M$*.*
- *There is a set of distinguished relations* $\{isLive_C : T_C \mid C \in M_{\mathsf{Class}}\} \subseteq \mathsf{Rel}_M$*.*
- *For each* $A \in M_{\mathsf{Association}}$*, such that* $A.\mathsf{ownedEnd.Property.type} = T_1$ *and* $A.\mathsf{memberEnd.Property.type} = T_2$*,* $A : T_1 \times T_2 \in \mathsf{Rel}_M$*.*
- *For every* $C \in M_{\mathsf{Class}}$ *and* $at \in C.\mathsf{ownedAttribute}$ *such that* $at.\mathsf{type} = T$*,* $at : T_C \times T \in \mathsf{Rel}_M$*.*

Example 2. The signature for the metamodel M_1 of Fig. 2(a) takes the following form, for $C \in \{Component, Composite, Attribute\}$:

$$\begin{aligned} Sig(M_1) \; = \; \langle \; & \{T_C, String\}, \; \{isLive_C : T_C, \; of : T_{Component} \times T_{Composite}, \\ & att : T_{Composite} \times T_{Attribute}, \; catt : T_{Component} \times T_{Attribute}, \; name : T_C \times String\}, \\ & OP_{String} \; \rangle \end{aligned}$$

Remark 1. Observe that both attributes and associations are formalized in the same way. An attribute of a metaclass is understood as a relationship that holds between an element of the metaclass sort and elements of the data type sort. Sorts $T_C \in \mathsf{Sort}_M$ are intended to denote the range of metaclasses for a given metamodel M. As we shall see, their semantics is taken to range over an infinite domain of possible instantiating metaobjects. However, every given metamodel instance contains a finite number of metaobjects. The predicate $isLive_C$ is consequently intended to always have a finite interpretation, denoting the set of the metaobjects of metaclass C that are operational or are alive in the metamodel instance. Note that multiplicities other than 1 or $*$ (including important kinds of multiplicities such as ranges) are not dealt with through the signature. Instead, these will be treated in the same way as metamodel constraints, as part of a larger, logical metamodel specification, defined next. Finally, subclassing can be understood as a subset relation among the live objects and inheritance can be imposed by suitable axioms. We have not considered this issue here, to focus on our constructive logical approach.

Before defining our logic, we first formulate a *value-based* semantics for metamodel signatures $Sig(M)$, based on the usual notion of $Sig(M)$-interpretation.

Definition 2 (Values). *Let* $T \in \mathsf{Sort}_M$*. The set of* values *of* T*, denoted by* $dom(T)$*, is defined as follows: if* T *is a data type, then* $dom(T)$ *is the set of values inhabiting it; if* T *is the type of a class* C*, then* $dom(T) = oid(C)$*, where* $oid(C)$ *is the set of object identifiers of class* C*.*

[2] To avoid confusion with other uses of the concept of a Type in the MOF, we use the term "sort" within our formalisation to denote classifying sets.

Note that here we assume that data type *values* are represented by ground terms of $Sig(M)$ and that oids are constants of $Sig(M)$. Values are the same in all the possible metamodel instances. Specific metamodel instances differ according to their representation of specific interpretations of the predicates $isLive_C$, An and at.

Definition 3 (Metamodel interpretation). *Take any metamodel M with signature $Sig(M)$. A* metamodel interpretation *is a $Sig(M)$-interpretation* $\mathbf{m}$ *such that:*

1. *sorts are interpreted according to Definition 2 and data type relations and operations are interpreted according to the implemented data types.*
2. *Each predicate $isLive_C$ is interpreted as a finite sub-domain $\mathbf{m}(isLive_C) \subset dom(T_C)$; intuitively, $\mathbf{m}(isLive_C)$ contains the metaobjects of class C that constitute* $\mathbf{m}$.
3. *Each association $An : T_1 \times T_2$ is interpreted as a relation $\mathbf{m}(An) \subseteq \mathbf{m}(isLive_{T_1}) \times \mathbf{m}(isLive_{T_2})$.*
4. *Each attribute $at : T_C \times T$ is interpreted as a functional relation $\mathbf{m}(at) \subseteq \mathbf{m}(isLive_{T_C}) \times dom(T)$.*

We treat the interpretation of sorts and data types in $\mathbf{m}$ as predefined, i.e., independently of the specific metamodel instance. The latter can be reconstructed from the interpretation of $isLive_C$, An and at. We represent this information with the model-theoretic notion of *diagram*:

Definition 4 (Diagram). *Given an interpretation* $\mathbf{m}$ *and values $v_1, \dots, v_n$, a* diagram Δ of a relation *r is the set of closed atomic formulas $r(v_1, \dots, v_n)$ that are true in* $\mathbf{m}$.

We will use diagrams as a *canonical representation of metamodel instances.*

Example 3. The metamodel instance I_1 of the metamodel M_1 in Figure 2 (d) is represented by the diagram:

$$\begin{aligned}\Delta_{M_1} = \{\ & isLive_{Composite}(cs),\ isLive_{Component}(ct),\ isLive_{Attribute}(sn),\ isLive_{Attribute}(na),\\ & of(ct,cs),\ att(cs,sn),\ catt(ct,na),\ name(cs,\texttt{"Family"}),\ name(ct,\texttt{"Person"}),\\ & name(sn,\texttt{"last\_name"}),\ name(na,\texttt{"first\_name"})\}\end{aligned}$$

2.2 Encoding the constraints of metamodels

Signatures formalize the metamodel structure and diagrams formalize the possible instantiations of this structure. However, a metamodel is not just a structure: it also includes constraints. A diagram is correct if it represents an instantiation that satisfies the constraints. How do we formalize a metamodel with constraints and the notion of correct diagram? That is, when can we say that all the information contained in the relational assertions of a diagram yields an actual metamodel instantiation, in the sense of conforming to the structure of the metamodel and its associated constraints?

We do this via a form of constructive logic. Essentially, we define a *realizability* relationship [18] between a logical formula F and what we call an *information term* τ (see [6] for more details), denoted $\tau : F$, yielding a notion of "information content" $\text{IC}(\tau : F)$, both to be defined next. Roughly, we will use $\tau : F$ to construct instance diagrams and $\text{IC}(\tau : F)$ to validate them. The realizability relationship forms a kind of type inhabitation relationship that is sufficiently powerful to include constraint satisfaction, essential for developing formal notions of provably correct instantiating model.

Definition 5 (Information terms and content). *Given a metamodel signature $Sig(M)$, the set of* information terms τ, (well-formed) formulas F *over Sig(M) and* information content $\textsc{ic}(\tau : F)$ *are defined as follows:*

`term` τ	`formula` F	$\textsc{ic}(\tau : F)$
`t`	$true(K)$	$\{K\}$
$\langle \tau_1, \tau_2 \rangle$	$F_1 \wedge F_2$	$\textsc{ic}(\tau_1 : F_1) \cup \textsc{ic}(\tau_2 : F_2)$
$j_k(\tau_k)$	$F_1 \vee F_2$	$\textsc{ic}(\tau_k : F_k) \quad (k = 1,2)$
$(v_1 \mapsto \tau_1, \ldots, v_n \mapsto \tau_n)$	$\forall x \in \{y : T \mid G(y)\}.F$	$\{\{y : T \mid G(y)\} = \{v_1, \ldots, v_n\}\} \cup \bigcup_{i=1}^{n} \textsc{ic}(\tau_i : F[v_i/x])$
$e(v, \tau)$	$\exists x : T.F$	$\textsc{ic}(\tau : F[v/x])$

where `t` *is a constant, K any first-order formula, $v \in dom(T)$, $\{v_1, \ldots, v_n\}$ is a finite subset of $dom(T)$ and G is a* generator*, namely a special formula true over a finite domain.*

Remark 2. Although the intuitive meaning of each of the formulas should be clear, some explanations are required for $true(\cdot)$ and for bounded universal quantification. The predicate $true(K)$ designs a metalogical assertion that the formula K is known to be true and that K, used as an assertion about a metamodel, does not require any specific information to be used in the construction of a metamodel instance (the idea was originally introduced in [12]). The information content is K: i.e., K is the minimal assumption needed to prove K. Regarding universal quantification, the associated information term defines both the domain $\{y : T \mid G(y)\} = \{v_1, \ldots, v_n\}$ and a map from this domain to information terms for F. In a metamodel signature, the formulas $isLive_C(x)$ is an example of generator. This corresponds to the assumption that a metamodel instance contains finitely many metaobjects.

If we consider universally bounded quantification $\forall x \in \{y : T \mid G(y)\}.F$ as semantically equivalent to $\forall x : T.G(x) \rightarrow F$ and $true(K)$ as equivalent to K, our formulas can be viewed as a subset of the ordinary (many sorted) first-order formulas and $\mathbf{m} \models F$ can be defined as usual. Furthermore, we may use any first-order formula as an argument of $true$. We will use formulas F to formalize constraints over metamodels and information terms to guarantee that F contains all the information needed to construct *valid* diagrams, i.e. if it is the diagram of a metamodel interpretation $\mathbf{m}$ that satisfies the constraints – we shall formalize this in Def. 7. We will represent a metamodel M by a $Sig(M)$-formula $Spec(M)$, where the latter encodes the constraints of the metamodel M so that if $\textsc{ic}(\tau : Spec(M))$ is (model-theoretically) consistent then the corresponding diagram is valid.

Consequently, with the aim of testing a transformation Tr, with input metamodel M_1 and output metamodel M_2, if we can generate a set of $\tau_j : Spec(M_1)$ with valid information content, we can generate the corresponding diagrams, which may then be fed into the transformation as test cases. Furthermore, if we specify the precondition of Tr as $true(Pre_{Tr})$ and its postcondition as $true(Post_{Tr})$, then we can use $Spec(M_1)$, $Spec(M_2)$, $true(Pre_{Tr})$ and $true(Post_{Tr})$ to check the correctness of Tr, i.e., to supply a test oracle.

We now define $Spec(M)$ and introduce some notation: for an atom B, $B(s^*)$ is an abbreviation of $\forall y : T.B(y) \leftrightarrow y \in s$ and $B(!x)$ an abbreviation of $\forall y : T.B(y) \leftrightarrow y = x$.

Definition 6 (Metamodel specification). *Given a metamodel signature $Sig(M)$, we represent each class C of M by a formula of the following form, where square brackets* *indicate optional sub-formulae:*

$$\begin{aligned} Spec(C) \;=\; & \forall x \in \{y : T_C \mid isLive_C(y)\}\,. \\ & \quad \exists z_1 : Set(T_{C_1})\,.\,true(A_1(x, z_1^*)\ [\wedge K_1(x, z_1)]) \wedge \cdots \wedge \\ & \quad\; \exists z_m : Set(T_{C_m})\,.\,true(A_m(x, z_m^*)\ [\wedge K_m(x, z_m)]) \wedge \\ & \quad\;\; \exists v_1 : T_{at_1}\,.\,true(at_1(x, !v_1)) \wedge \cdots \wedge \\ & \quad\;\;\; \exists v_n : T_{at_n}\,.\,true(at_n(x, !v_n)\ [\wedge true(K_C)]) \end{aligned}$$

1. *$A_i : T_C \times T_{C_i}$ $(0 \le i \le m)$ are the relations of* Rel_M *corresponding to the associations;*
2. *K_i $(0 \le i \le m)$ are "multiplicity constraints" on the associations A_i;*
3. *$at_j : T_C \times T_{at_j}$ $(0 \le j \le n)$ are the relations of* Rel_M *corresponding to the attributes of C;*
4. *K_C is a formula that we call the "class constraint" of C.*

A metamodel M is specified by the $Sig(M)$-formula

$$Spec(M) \;=\; Spec(C_1) \wedge \cdots \wedge Spec(C_l)\ [\wedge\, true(K_G)] \qquad (1)$$

where $C_1, \ldots, C_l$ are the metaclasses of M and K_G is an optional formula that we call the "global constraint" of M.

We switch to a "light notation", where we use generators $isLive_C$ as sorts.

Example 4. The specification of the metamodel M_1 described in Example 2 is $Spec(M_1) = F_{Cmt} \wedge F_{Cms} \wedge F_{Att} \wedge true(K)$, where:

$$\begin{aligned} F_{Cmt} \;=\; & \forall x : isLive_{Component}(x)\,. \\ & \quad \exists sc : Set(Composite)\,.\,true(of(x, sc^*) \wedge size(sc) = 1) \wedge \\ & \quad\; \exists sa : Set(Attribute)\,.\,true(catt(x, sa^*) \wedge \forall a \in sa\,.\,catt(!x, a)) \wedge \\ & \quad\;\; \exists s : String\,.\,true(name(x, !s) \wedge cattNames(x, s)) \\ F_{Cms} \;=\; & \forall x : isLive_{Composite}(x)\,. \\ & \quad \exists sa : Set(Attribute)\,.\,true(att(x, sa^*) \wedge \forall a \in sa\,.\,att(!x, a)) \wedge \\ & \quad\; \exists s : String\,.\,true(name(x, !s) \wedge attNames(x, s)) \\ F_{Att} \;=\; & \forall x : isLive_{Attribute}(x)\,.\,\exists s : String\,.\,true(name(x, !s)) \end{aligned}$$

We omit the specification of K for conciseness; informally, it states that different attributes linked to the same component/composite must have different names. In F_{Cmt}, the constraint $size(sc) = 1$ encodes the multiplicity 1 of the target association end of `of`, while $\forall a \in sa\,.\,catt(!x, a)$ refers to the multiplicities 0..1 of the source association ends of `catt`. The encoding of multiplicity constraints is standard, i.e., it can be fully automated. The other constraints are encoded in Prolog, for reasons that will be explained at the end of the section. For example, the constraint $cattNames(x, s)$ is defined by the clause:

$$\texttt{false} \leftarrow isLive_{Attribute}(a) \wedge catt(x, a) \wedge name(x, s)$$

i.e., it is false that there is an attribute a linked to component x whose name is s; the constraint $attNames(x, s)$ is similar.

Let $\tau : Spec(M)$ be an information term for a specification of a metamodel M. We distinguish between a "*diagram part*" IC_D and a "*constraint part*" IC_C of the information content. The latter contains the multiplicity constraints, the class constraints and the global constraint, while the former contains the *domain formulas* $isLive_C(o_i)$ for $\{y : T_C | isLive_C(y)\} = \{o_1, \ldots, o_n\}$, the *association formulas* $An(o, s^*)$ and the *attribute formulas* $at(o, !v)$. The diagram part allows us to define a bijective map δ from the information terms for $Spec(M)$ into the instance diagrams for M, by means of the following conditions:

1. Domain formulas: $isLive_C(o_i) \in \delta(\tau)$ iff $isLive_C(o_i) \in \text{IC}_D(\tau : Spec(M))$;
2. Association formulas: $An(o, o') \in \delta(\tau)$ iff $An(o, s^*) \in \text{IC}_D(\tau : Spec(M))$ and $o' \in s$;
3. Attribute formulas: $at(o, d) \in \delta(\tau)$ iff $at(o, !d) \in \text{IC}_D(\tau : Spec(M))$.

Example 5. Consider the specification $Spec(M_1)$ of Example 4 and the information term $\tau_1 = \langle \tau_{1_1}, \tau_{1_2}, \tau_{1_3}, \mathtt{t} \rangle$, where:

$$\begin{aligned}
\tau_{1_1} : F_{Cmt} &= (ct \mapsto e(\{cs\}, \langle \mathtt{t}, e(\{na\}, \langle \mathtt{t}, e(\texttt{"Person"}, \mathtt{t}) \rangle) \rangle)) \\
\tau_{1_2} : F_{Cms} &= (cs \mapsto e(\{sn\}, \langle \mathtt{t}, e(\texttt{"Family"}, \mathtt{t}) \rangle)) \\
\tau_{1_3} : F_{Att} &= (sn \mapsto e(\texttt{"last\_name"}, \mathtt{t}),\ na \mapsto e(\texttt{"first\_name"}, \mathtt{t}))
\end{aligned}$$

The diagram and constraint part of $\text{IC}(\tau_1 : Spec(M_1))$ are:

$$\begin{aligned}
IC_D = \{ & isLive_{Component}(ct),\ isLive_{Composite}(cs),\ isLive_{Attribute}(sn),\ isLive_{Attribute}(na), \\
& of(ct, \{cs\}^*),\ catt(ct, \{na\}^*),\ att(cs, \{sn\}^*),\ name(ct, !\texttt{"Person"}), \\
& name(cs, !\texttt{"Family"}),\ name(sn, !\texttt{"last\_name"}),\ name(na, !\texttt{"first\_name"}) \} \\
IC_C = \{ & size(\{cs\}) = 1,\ \forall a \in \{na\}.catt(!ct, a),\ \forall a \in \{sn\}.att(!cs, a), \\
& cattNames(ct, \texttt{"Person"}),\ attNames(cs, \texttt{"Family"}) \}
\end{aligned}$$

The diagram $\delta(\tau_1)$ coincides with Δ_{M_1} of Example 3. Clearly, we could start from Δ, reconstruct IC_D and, from the latter, reconstruct τ_1. We can consider the diagram part as the "encoding" of the pure UML part of an UML model, as depicted in part (b) of Fig. 2.

IC_C does not play any role in the definition of the map δ, but encodes the constraints. In the above example, the constraint part is satisfied: $size(\{cs\}) = 1$ is true and one can easily see that the other formulas in IC_C are also true. We will say that τ_1 *satisfies the constraints*.

Definition 7 (Constraint satisfaction). *Let $\tau : Spec(M)$ be an information term for a specification of a metamodel M: we say that τ (and $\delta(\tau)$)* satisfies the constraints *iff $\Delta = \delta(\tau)$ is the diagram of a metamodel instance $\mathbf{m}_\Delta$ of M such that $\mathbf{m}_\Delta \models \text{IC}_C(\tau : Spec(M))$.*

Let $\mathbf{m}_\Delta$ be as in the above definition. We can prove that $\mathbf{m}_\Delta \models \text{IC}(\tau : Spec(M))$ hence the valid metamodel instances of M are models (in the sense of logic) of $Spec(M)$. Finally, the following sufficient condition for satisfying the constraints can be proven:

Theorem 1. *Let $\tau : Spec(M)$ be an information term for a specification of a metamodel M, Ax any set of axioms that are true over all the metamodel-instances and $K = \bigwedge \text{IC}_C(\tau : Spec(M))$. Then:*

a) if $Ax \cup \mathrm{IC}_D(\tau : Spec(M)) \vdash K$*, then* τ *(and* $\delta(\tau)$*) satisfies the constraints;*
b) if $Ax \cup \mathrm{IC}_D(\tau : Spec(M)) \vdash \neg K$*, then* τ *(and* $\delta(\tau)$*) does not satisfy the constraints.*

In *Ax* we have the axioms for the data types (integers, strings, sets, ...) and the general assumptions on the oids and on the metamodel instances. Assume we have a *possible* term $\tau : Spec(M)$. To establish that the diagram $\delta(\tau)$ is a valid metamodel instance we could apply Theorem 1 and we could attempt to derive a proof using a suitable inference system for *Ax*; alas, this is hardly predictable if *Ax* and constraints are full first-order formulae. We need a tractable constraint language. The constraints concerning multiplicities are recognized and checked in the generation phase. To express and check the other constraints, we rely on Horn clauses. We distinguish between problem domain and absurdity clauses. The former are definite clauses implementing data types (strings, sets, bags, ...) and encoding the general properties of the MOF. The latter have the form `false` $\leftarrow Body$, with intended meaning $\neg\exists\mathbf{x}\,.\,Body$. By the properties of definite programs, one can prove:

a) if `false` finitely fails, then for every absurdity constraint `false` $\leftarrow Body$, $\neg\exists\mathbf{x}\,.\,Body$ is satisfied;
b) if `false` succeeds, then $\neg\exists\mathbf{x}\,.\,Body$ is falsified for some constraint.

For example, the constraint part of the information term of Example 5 contains two absurdity constraints, namely $cattNames(ct,$ `"Person"`$)$ and $attNames(cs,$ `"Family"`$)$ and `false` finitely fails, since there is no attribute with name `"Person"` or `"Family"`. Had the diagram contained such an attribute, `false` would had succeeded.

3 Testing via model generation

In this Section we show how our setup allows us to:

1. generate input test models that meet the precondition for a model transformation;
2. check that, after the input models have been transformed, the transformation postcondition is met.

In this way, we are able to provide a constructive model transformation testing framework in accord with the concepts of Fig. 1.

The System. The architecture of our system is outlined in Fig. 3 with two main units (*Test Case Generator* and *Oracle*) cooperating. The module `testGen` generates a set of information terms that are translated into test cases for the transformation undergoing testing (*Tr* U.T.) by the module `toMOF`. The inputs of `testGen` are the specification $Spec(M_1)$ of the source metamodel M_1, the transformation specifications precondition, encoded by a constraint Pre_{Tr} and a set *GR* of *generation requests* provided by the user to implement *test criteria*. The generated information terms $\tau_k : Spec(M_1) \wedge true(Pre_{Tr})$ are translated into test cases $I_k = \mathtt{toMOF}(\tau_k)$. The transformation *Tr* U.T. is then run using such test cases. Note that we are *agnostic* about how models are represented — as abstracted in the `fromMM` module — as well as *Tr* U.T. is implemented: the framework could translate from arbitrary metamodelling language and feed test cases into a transformation written in any language. However, the transformation can also be assumed to be written in the *same* constructive language as the metamodel specification and information terms (together with the modules themselves) because the logic may rely on a

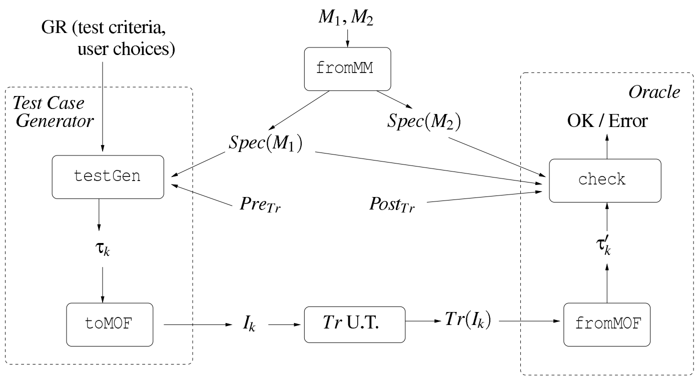


Fig. 3. Testing transformations

λ-calculus rather than realizability. The reader is referred to [14] for details of such an encoding.

The translated metamodel instances are checked using the module `fromMOF`, which attempts to reconstruct the information terms corresponding to the transformation output model $Tr(I_k)$ and the `check` module, which tries to validate $Post_{Tr}$. The two modules work together as a test *oracle*: if `fromMOF` fails, then the results do not respect the types or the invariants prescribed by $Spec(M_2)$, while if `check` fails, then only $Post_{Tr}$ is violated.

The `testGen` module supports *exhaustive* (EGen) and *random* generation (RGen). In RGen, objects, attribute values and links are generated randomly, albeit consistently with the *GR* and the constraints. EGen, instead, generates all consistent information terms, which are *minimal* w.r.t. the constraints and a given population As emphasized in [7], the generation of "small test-cases" is crucial to make the understanding of each test case easier and to allow an efficient diagnosis in presence of errors. Finally, *data-centric* test criteria such as those considered in *op. cit.* can be expressed as generation requests.

Before delving into the specifics of the generation algorithm behind `testGen`, we illustrate how the architecture would operate over the source metamodel M_1 and the target metamodel M_2 shown in Fig. 2.

Example 6. The specification $Spec(M_1)$ is the one of Example 4, while $Spec(M_2) = F_{Tab} \wedge F_{Col} \wedge K_{Tab}$, where:

$$\begin{aligned} F_{Tab} &= \forall t : isLive_{Table}(t)\,. \\ &\qquad \exists cl : Set(Column)\,.\,true(with(t, cl^*)) \wedge \exists s : String\,.\,true(id(t, !s) \wedge id(!t, s)) \\ F_{Col} &= \forall cl : isLive_{Column}(cl)\,.\,\exists s : String\,.\,true(id(cl, !s)) \end{aligned}$$

and the constraint K_{Tab} states that the `id`'s occurring in a table are pairwise distinct. Let Tr be a transformation mapping M_1-instances into M_2-instances informally defined as follows:

- *Pre-condition.* Each `Composite` object is linked to a unique `Attribute` via `att`.
- *Post-condition.* For every `Composite` object *cs* and every `Component` *ct* `of` *cs* there is a corresponding `Table` *t* such that:

- t.`id` is equal to ct.`name`;
- for every `Attribute` a linked to ct by `catt` there is a corresponding `Column` cl linked to t by `with`, so that cl.`id` is equal to a.`name`;
- for every `Attribute` a linked to cs by `att` there is a `Column` cl linked to t by `with` such that cl.`id` is equal to a.`name`.

For example, the metamodel instance I_1 of M_1 is transformed into $I_2 = Tr(I_1)$ of M_2 (see Fig. 2). The formal specification of Tr contains formulas such as:

$$\begin{array}{lll} toTable(ct,t) & \leftrightarrow & isLive_{Component}(ct) \wedge isLive_{Table}(t) \wedge \exists s : String.name(ct,s) \wedge id(t,s) \\ \texttt{false} & \leftarrow & isLive_{Component}(ct) \wedge \neg(\exists t : isLive_{Table}(t).toTable(ct,t)) \end{array}$$

For the experiments discussed in Table 1, we have implemented Tr in Prolog. To generate the test cases, we supplied (beside $Spec(M_1)$ and the precondition) the following generation requests, whose precise syntax we omit for the sake of space:

- *PR* (Population Requests): generate at most 2 composites, exactly 2 components and at most 8 attributes.
- *MR* (Multiplicity Requests): a snapshot must contain at most 2 components linked to a composite (via `of`) and at most 2 attributes linked to a component (via `catt`). No MR is needed for the association ends with multiplicity `1` or `0..1`.
- *AR* (Attribute Requests). These are: (AR_a) all the objects have distinct `name`; (AR_b) there are at least two `Attribute` with the same `name`.

Experiment	Module	Input	Result	Time (sec.)
1a	Test Case Generator	$Spec(M_1)$, Pre_{Tr}, *PR*, *MR*, AR_a	18 test cases I_k	0.7
	Oracle	18 translations $Tr(I_k)$	9 failed	0.31
1b	Test Case Generator	$Spec(M_1)$, Pre_{Tr}, *PR*, *MR*, AR_b	118 test cases I_k	5.9
	Oracle	118 translations $Tr(I_k)$	76 failed	1.9
2	testGen	$Spec(M^*)$, GR^*, Pre^*	144 τ_k^*	4.4

Table 1. Experimental results

We use those requests to deal with *coverage* criteria such as those introduced in [7], adapting previous work on test adequacy criteria for UML. For example, "Association-end-Multiplicities" (a test case for each representative multiplicity pair) can be implemented by MR's, whereas "Class Attribute" (a test case for each representative attribute value) by AR's. In our experiment we have used *partition analysis* of the input domain, for example into (1a) the snapshots where all the objects have distinct names and (1b) those containing at least two distinct attributes with the same name. For (1a) and (1b) we have relied on EGen, with the above PR to avoid a combinatory explosion of the number of minimal snapshots.

The experiments have been conducted with an Intel 2.2 GHz processor T7500 operating with 2GB of RAM and the results are summarized in Table 1. EGen for (1a) has been performed according to the generation requests *PR*, *MR* and AR_a. We have obtained 18 test cases. Running Tr and the oracle, 9 of the 18 test cases failed. EGen for (1b) used *PR*, *MR* and AR_b and yielded 118 test cases. Again we ran Tr and the oracle; 76 test cases failed. As illustrated by the sample test cases in Fig. 4, two kinds of errors were detected:

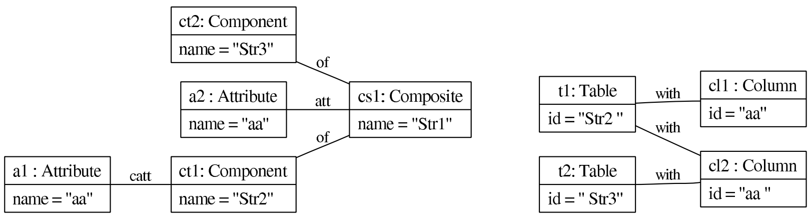


Fig. 4. Failed test cases

1. The column `cl2` corresponding to `a2` occurs in both the tables `t1` and `t2` corresponding to `ct1` and `ct2` resp., i.e., the multiplicity 1 of `with.Table` is violated.
2. The table `t1` corresponding to `ct1` contains two columns with `id = "aa"`.

Error 1 was reported in all 9 failed test cases of (1a) and can be traced back to a wrong implementation of Tr. Error 2 can be discovered only by (1b) and requires a stronger precondition. Without exhaustive generation for the (1a, 1b)-partition, the above errors could not have been revealed. We believe that the combination of exhaustive generation of the *minimal* information terms with domain partition testing, using the latter to avoid combinatory explosion, could be profitable line of research.

In the `testGen` module, the information terms are generated in two phases. In the first one, some existential parameters are left *generic* (this is possible thanks to the constructive structuring of the information). In the second phase, the generic parameters are instantiated according to a strategy, e.g. RGen/EGen. As a rule of thumb, one leaves generic those existential parameters that give rise to a combinatory explosion of the number of solutions. In particular, in the first phase of experiments (1a, 1b) we used *PR* and *MR* leaving `name` open; we obtained 18 generic solutions in 0.01 sec. In the second phase of (1b), the names were instantiated according to AR_b (at least two attributes with name `"aa"` and the other ones distinct): we obtained 118 test cases. As another larger example, we have formalized the UML metamodel considered in [16] (denoted with star in the table) and we have generated 144 generic terms in 4.4 sec.

4 Related work and conclusions

The relevance of snapshot generation (SG) for validation and testing in OO software development is widely acknowledged and has yielded several animation and validation tools supporting different languages. USE [8] has been the first system supporting automatic SG; differently from us, SG is achieved procedurally, requiring the user to write Pascal-like procedures in a dedicated language. Not coincidentally, USE performances are very sensitive to the *order* of objects and attribute assignments [2]. Alloy [9] is the leading system [2] for generation of instances of invariants, animation of the execution of operations and checking of user-specified properties. Alloy compiles a formula in first-order relational logic into quantifier-free booleans and feeds to a SAT solver. Alloy's original design was quite different from UML, but recent work [1] has brought the two together. Baudry et al. have developed a tool that maps metamodel descriptions in the Ecore framework into Alloy, delegating to the latter the generation of model snapshots [15]. Their approach is focused on generation of models solely from metamodel encoding. Our approach could be used to enhance their result, through our use of realizability to tighten the relationship among (meta)models and their transformations. On the other hand, they have extensively investigated the issue of the quality and adequacy of test models, both in term of generation strategies and of mutation analysis [15]. We have already adopted a version of their *domain partition* strategy to guide test generation and plan to refine it towards the filtering of *isomorphic* and thus useless test models.

Another close relative is FORMULA [10], a tool supporting a general framework for model-based development. Similarly to us, it is based on logic programming, more specifically model generation, employing abduction over non-recursive Horn logic with stratified negation. Model transformations are also encoded logically. Since abduction can be computationally expensive, we plan to compare it with our realizability-based SG, as the FORMULA setup may give us a hook to more general domain-specific modeling languages.

Our particular approach to snapshot generation builds on work by three of the present authors [6, 13] for object-oriented test case generation, but is now extended and combined with a first-order variant of the constructive encoding of the MOF and model transformations by the other author [14]. By combining the two approaches, we obtain the first universal encoding of its kind. Further, our logic supports the “proofs as programs” paradigm, where a constructive proof π of $TR \vdash F_1 \rightarrow F_2$ represents a program P_π mapping the realizers of F_1 into realizers of F_2, TR being a set of transformation formulae. Hence validation naturally leads to formal *certification* via proof assistants. This is indeed the object of future work.

A number of authors have attempted to provide a formal understanding of metamodelling and model transformations. We refer to [14] for a review. In particular, rule-based model transformations have a natural formalization in graph rewriting systems [11]: for example, Ehrig et al. have equipped graph grammars with a complex notion of instance generation, essentially adding a means of generation directly into graph rewriting [5]. As with our work, their formalism permits a uniform semantics of model transformations, specification and test generation, although with a fairly heavy mathematical overhead. However, their approach is by definition applicable within a rule-based paradigm: in contrast, because our tests are contractual and based in the very generic space of constructive logic, we need not restrict ourselves to rule-based transformations.

While full verification of model transformations can be as difficult to achieve as in ordinary programming, the power of model transformations demands some formal guarantee that any transformation algorithm actually produces the tests that we expect: in particular, that tests cases are of appropriate metamodel types and satisfy generation constraints/requests. We have developed a constructive encoding of the MOF that facilitates this via a uniform, single-language treatment of models, metamodels, instantiation, transformation specification, test case generation constraints *and* test cases generation. To the best of our knowledge, a similar approach has not been explored before. Our implementation relies on a fairly naive encoding of the MOF in logic and needs future work to be readily integrated into other tool sets: in particular, we need to investigate how standard visual representations of metamodels and transformations might complement the approach, together with an automatic translations from OCL into our logic for constraint representation. There are further optimisations that can be done to improve the constraint solver: for example, isomorphic solutions are reduced but, currently, not eliminated; divide and conquer strategies such as modularization of tests could be employed to overcome the potential combinatory explosion for large metamodels. Our approach can currently be considered as one way of integrating formal metamodelling perspectives with snapshot generation for testing. While formal metamodelling opens up the possibility of full transformation verification, from a practical perspective, testing is likely to remain an integral component of transformation development for the long term. However, by following an approach such as ours, formal metamodelling can still be exploited to generate test data in a way that is guaranteed to preserve consistency with required constraints. For this reason, we see this work as opening up a very promising line of research for the formal metamodelling community.

References

1. K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 9(1):69–86, 2010.
2. E. G. Aydal, M. Utting, and J. Woodcock. A comparison of state-based modelling tools for model validation. In R. F. Paige and B. Meyer, editors, *TOOLS (46)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 278–296. Springer, 2008.
3. B. Baudry, T. Dinh-Trong, J.-M. Mottu, D. Simmonds, R. France, S. Ghosh, F. Fleurey, and Y. L. Traon. Model transformation testing challenges. In *ECMDA workshop on Integration of Model Driven Development and Model Driven Testing*, 2006.
4. J. Bézivin, B. Rumpe, A. Schürr, and L. Tratt. Model transformations in practice workshop. In J.-M. Bruel, editor, *MoDELS Satellite Events*, volume 3844 of *LNCS*, pages 120–127. Springer, 2005.
5. K. Ehrig, J. M. Küster, and G. Taentzer. Generating instance models from meta models. *Software and System Modeling*, 8(4):479–500, 2009.
6. M. Ferrari, C. Fiorentini, A. Momigliano, and M. Ornaghi. Snapshot generation in a constructive object-oriented modeling language. In A. King, editor, *LOPSTR*, volume 4915 of *LNCS*, pages 169–184. Springer, 2007.
7. F. Fleury, J. Steel, and B. Baudry. Validation in model-driven engineering: Testing model transformations. In *MoDeVa'04 (Model Design and Validation Workshop associated to ISSRE'04)*, Rennes, France, November 2004.
8. M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL models in USE by automatic snapshot generation. *Software and System Modeling*, 4(4):386–398, 2005.
9. D. Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.
10. E. Jackson and J. Sztipanovits. Formalizing the structural semantics of domain-specific modeling languages. *Software and Systems Modeling*, 2009.
11. A. Königs and A. Schürr. Multi-domain integration with mof and extended triple graph grammars. In J. Bezivin and R. Heckel, editors, *Language Engineering for Model-Driven Software Development*, number 04101 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2005.
12. P. Miglioli, U. Moscato, M. Ornaghi, and G. Usberti. A constructivism based on classical truth. *Notre Dame Journal of Formal Logic*, 30(1):67–90, 1989.
13. M. Ornaghi, M. Benini, M. Ferrari, C. Fiorentini, and A. Momigliano. A constructive object oriented modeling language for information systems. *ENTCS*, 153(1):67–90, 2006.
14. I. Poernomo. Proofs-as-model-transformations. In A. Vallecillo, J. Gray, and A. Pierantonio, editors, *ICMT 2008, Proceedings*, volume 5063 of *LNCS*, pages 214–228. Springer, 2008.
15. S. Sen, B. Baudry, and J.-M. Mottu. On combining multi-formalism knowledge to select models for model transformation testing. In *ICST*, pages 328–337. IEEE Computer Society, 2008.
16. S. Sen, B. Baudry, and J.-M. Mottu. Automatic model generation strategies for model transformation testing. In R. F. Paige, editor, *ICMT*, volume 5563 of *LNCS*, pages 148–164. Springer, 2009.
17. Y. L. Traon, B. Baudry, and J.-M. Jezequel. Design by contract to improve software vigilance. *IEEE Trans. Softw. Eng.*, 32(8):571–586, 2006.
18. A. S. Troelstra. Realizability. In S. R. Buss, editor, *Handbook of Proof Theory*, chapter IV, pages 407–473. Elsevier, 1998.